A Language Processing Tool for Program Comprehension

Mario M. Berón
San Luis University, Department of Informatics
San Luis, Argentina, 5700
e-mail: mberon@unsl.edu.ar

Pedro R. Henriques
Minho University, Department of Informatics
Braga, Portugal
e-mail: prh@di.uminho.pt

Maria J. Varanda Pereira
Minho University, Department of Informatics
Braga, Portugal
e-mail: mjoao@ipb.pt

Roberto Uzal
San Luis University, Department of Informatics
San Luis, Argentina, 5700
e-mail: ruzal@sinectis.com

Germán Montejano
San Luis University, Department of Informatics
San Luis, Argentina, 5700
e-mail: gmonte@unsl.edu.ar

Abstract

Program Comprehension is a Software Engineering discipline which aims to understand computer code written in a high-level programming language. Program Comprehension is useful for reuse, inspection, maintenance, reverse engineering and many other activities in the context of Software Engineering.

In this paper we define a set of techniques to extract static and dynamic information from the target program. These techniques are based on the inclusion of inspection functions and control statements in the system's source code. The first are intended to show the functions actually used. The second are necessary to reduce the number of functions recovered for a better administration. We show a possible implementation of this approach using a language processor generator very useful and easy to use.

Our strong motivation was to support the understanding of routing algorithms, available in EAR a *routing algorithms evaluation system.* To assist the program comprehension task, we generate different views that use the information extracted by our strategy, such as *the routing algorithm output* (that can be seen as a *problem domain view*), or the sequence of called functions, and their source and object code (examples of *program domain views*).

Although specific, we intend to generalize this approach.

Keywords: Program Comprehension, Comprehension Software, System's views, Inspections functions.

1 INTRODUCTION

The faster world changes, the bigger is the need to rebuild software systems. Due to the complexity, programmers usually prefer to rewrite instead of reuse it. Program Comprehension (PC) intends to provide ways to help in that task.

Program comprehension [5][10][15][19] is a very challenging subfield of Software Engineering. The main idea is to provide program's embedded information in new representations and new techniques of information extraction.

In this paper, we define a technique—code annotation for program inspection—that allows us to build useful static and dynamic views for program understanding. This technique is thought to be applied initially to programs written in imperative language with a sequential model of computation with shared memory. A similar approach aiming at Object Oriented languages can be found in [12].

As a starting point to study the role of code annotation in PC, we selected a routing algorithm evaluation system, called EAR [9][1][2][3], that we have developed so far. The main output of this system is a graphical representation. Some of the algorithms (for example the geometric routing algorithms) included are novelties, but even those that are well known are very complex; they are very difficult to understand because they use mathematical concepts simple to explain but whose implementation requires many lines of code.

Our motivation was precisely to investigate and implement PC strategies to create software aids that help programmers to understand how those routing algorithms produce their output.

Furthermore, we believe that we will be able to generalize the results so far obtained to create program comprehension tools to support other systems.

The paper is organized as follows. Section 2, defines and classifies comprehension software. Section 3, defines techniques to extract information from programs. Section 4, shows a possible way to implement these techniques. Section 5, describes the application of our strategies to the routing system [4][6][9]. Finally, we present the conclusion and future work.

2 PROGRAM COMPREHENSION TOOLS

A *Program Comprehension System (PCS)* [11][14][15][16][19] is a set of related programs that aims at making the understanding of a software system easier through the presentation of different perspectives or views. Basically a PC Tool has modules: to parse and analyze the target code (the programs that compose the software system under study); to store and handle the knowledge extracted; to visualize all the retrieved information. The ability to offer different views of that knowledge and the interactivity of the interface (allowing for a complete navigation over the extracted information) are the main characteristics of those PC tools that should offer very abstract, high-level, views of the system as well as detailed, low-level, ones.

PC Software can be classified considering the strategies used or their scope. Concerning strategies, the most used are *top-down, bottom-up,* or *hybrids*; considering the application range, we can

classify the PC Tools in two very important classes: *general purpose*, and *domain specific* tools. A survey and ranking of comprehension tools can be found in these paper [20][18][17][7] [8]. Our tool, discussed along the paper is clearly in the second class.

More precisely, we are interested in building comprehension tools with a specific purpose, using both *static and dynamic strategies* [7][18], and adopting the idea of Brooks [10] of offering simultaneously a behavioral (problem domain) view, and an operational one (program domain). Static strategies consist of exploration techniques that build a visual representation of the components of a software system and their relationships, as well as its data and control flows. Dynamic strategies are based on runtime program analysis, and are intended to show the system behavior during its execution.

3 SOURCE CODE INSPECTION BY ANNOTATION

In this section, we discuss a technique, called source code annotation, that allow us to get the functions execution sequence and control the program's execution flow.

This task is carried out inserting automatically *inspection functions* in the source code of the system. We think that the interesting check points are the *beginning* and the *end* of the each function, because we want to detect which functions are actually invoked by the system during execution.

Each time the system begins the execution it invokes its functions. The first statement executed by each function invoked is the *inspector function call*. This inspector can initially print the name of the executed function and furthermore indicate the beginning of its execution. When the function finalizes, the inspector inserted at the end must inform this event; actually, it can report other complementary information that will not be described here. Although not strictly necessary, that exit inspector is included to control recursion. In this way we can trace the program building and showing the execution path. We believe that displaying the name of the functions and their invocation order will help the programmer to understand the algorithms behavior.

Figure 1.a shows a normal function definition, and Figure 1.b. illustrates the code transformation after the insertion of both start and exit inspectors.

```
int f (int x, int y)
        {       float z,y;
                /* more declarations*/
                ……
                /* actions */
                return value
        }
```

(a)

```
int f (int x, int y)
        {       float z,y;
                /* more declarations*/
                INPUT_INPECTOR("f");
                ……
                …….
                /* actions */
                OUTPUT_INSPECTOR("f")
                return value;
        }
```

(b)

Figure 1: Insertion of inspection functions

This is the way we obtain the functions' execution flow. In practice, we face a problem concerned with the number of invocations that can be too large; displaying all the calls can be too messy and

not helpful. For instance, one situation where that problem can emerge is within iterations. That is the reason we need to control the number of invoked functions we want to display. In order to overcome this problem, we insert code *before*, *within* and *after* the iteration statement; observe Figure 2.

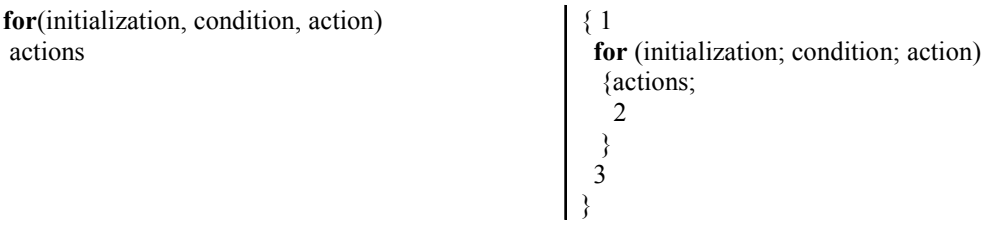

Figure 2: Insertion Scheme

We distinguish three main points, identified in Figure 2 by numbers 1, 2 and 3. In 1 and 3, we insert a control function *showFunction*(*value*) that pushes/pops the control value onto the control stack. In 2, we insert other control function called *dec*() that decrements the value found in the stack top. If that value is zero, the start and exit inspectors do not show the function's information.

4 IMPLEMENTATION OF SOURCE CODE ANNOTATIONS

To implement the techniques explained in section 3 we use attribute grammars and the Lex and Yacc tools. We built a scanner and parser of ANSI-C and then we incorporated the attributes as well as necessary semantic actions to accomplish the objective.

We use the following attributes:

- *Code*: synthesize the system source code.
- *Return numbers*: synthesize the number of return statements of a function.
- *Caller function name*: synthesizes the function name defined in the module.
- *Is within a function*: used to determinate the beginning of each function.
- *Function type*: contains the function return type.
- *Line number*: indicates the line number of each sentence in the source code.
- *Function name*: stores the function's name.

The Yacc specifications refer to these attributes concatenating the words that constitute their names. We know that the use of inherited attributes in Yacc is not allowed. For this reason, we choose to implement the inherited attributes using global variables. Since there are few inherited attributes, then we need few global variables and we obtain a Yacc code readable and easy to study. In the following subsections we explain the specifications for each operation.

4.1 Synthesis and Code Insertion

The synthesis and code insertion is made by using the attributes *code*, *function type*, *is within a function* and *function name*. The task must be done carefully without altering the program's semantic. At the beginning of the functions these jobs are simple. If the function is empty we insert code after open brace. If the function has only data declaration, then we insert code after data declaration. If the function has only statements then we insert code before the first sentence. Finally, if the function has both data and statements we insert code after data. An example of these operations is shown in Figure 3.

```
compound_statement : ….
  | '{' declaration_list
     {if (isWithinFunction==TRUE)
      {insertInputInspectionFunction($<data>2.code,functionName); isWithinFunction=FALSE;}}
     statement_list  '}' { copy($<data>$.code,"{");  concat($<data>$.code,$<data>2.code);
                           concat($<data>$.code,$<data>4.code);      concat($<data>$.code,"}");    }
```

Figure 3: Synthesis and Insertion Code

The attributes *is within function* and *function name* are initialized when the parser finds one function definition. We emphasized this operation in Figure 4.

```
function_definition :……
    | declaration_specifiers declarator
       {isWithinFunction=TRUE;
        strcpy(functionName,$<datas>2.callerFunctionName);
        ……..
       } compound_statement {……}
```

Figure 4: Initialization of attribute *is within function*

When the attribute *is within function* is true the function *insertInputInspectorFunction*-(*code*,*functionName*) inserts the function *INPUT_INSPECTOR*(*functionName*) in the source code. Then the attribute *is within function* is initialized as false. This operation is necessary because we need to avoid inserting code in other compound statements.

The most interesting case appears when we want to insert code at the end of the functions. In these cases, we need to insert a composed statement. The statement contains the declaration of one variable which type is the type of returned value for the study function. This variable must be initialized with the value of the expression contained in the return statement. This procedure is necessary because within an expression used in a return statement may have invocation to other functions. In these situations, we use the attribute *function type*. This attribute is inherited because the function type is available before processing function's name and its body, and it is initialized each time that the parser finds a function definition. Therefore this value must be transmitted to the non-terminal compound sentence in the production that specifies a function. The Figure 5 shows the code transformation made and the Figure 6 shows the corresponding Yacc specification.

Note that this approximation eliminates the problem of the program semantic modification when the return statements are the only statements within iteration and selection statements (see Figure 7).

```
struct data f (int x, ……)
        {Declarations
         Statements
         return v;
        }
```

```
struct data f (int x, ……)
        {Declarations
         INPUT_INSPECTOR("f");
         Statements
         { struct data nruter;
           nruter=v;
           OUTPUT_INSPECTOR("f");
           return nruter;
         }
        }
```

Figure 5: Result of synthesis scheme

```
jump_statement: ……
  | RETURN ';'
    {…… $<data>$.haveReturn=1; insertOutputInspector($<data>$.code,functionName);   }
  | RETURN expression ';'
    {…… $<data>$.tieneReturn=1;          copy($<data>$.code,"{ ");
     concat($<data>$.code,functionType);  concat($<data>$.code," nruter;");
     concat($<data>$.code," nruter = ");  concat($<data>$.code,$<data>2.code);
     concat($<data>$.code,";");           insertOutputInspector($<data>$.code,functionName);
    }
```

Figure 6: Insertion of return sentence

```
int f (int x, …..)
  {Declarations  …….
   if (condition)
       return value;
       else
         return otherValue;
  }
```

```
int f (int x,…..)
  {Declarations  ……
   if (condition)
      {int nruter;
         nruter= value;  return nruter;
      } else
        {int nruter;
            nruter=otherValue;  return nruter;
        }
  }
```

Figure 7: Strategy apply to selection statements

4.2 Iteration Control

The incorporation of the iteration control statements can be done when we synthesized the source code. Each time the parser finds an iteration statement it incorporates the scheme for iteration control described in section 4. The Yacc specification for that task can be seen in Figure 8.

```
iteration_statement:
  ……
  |FOR '(' expression_statement expression_statement expression ')'
    statement
    {……
     copy($<data>$.code,"{ showFunction(0);");  concat($<data>$.code,$<data>1.value);
     concat($<data>$.code,"(");                 concat($<data>$.code,$<data>3.code);
     concat($<data>$.code,$<data>4.code);       concat($<data>$.code,$<data>5.code);
     concat($<data>$.code,") {");               concat($<data>$.code,$<data>7.code);
     concat($<data>$.code,"dec( ); }");         concat($<data>$.code,"showFunction(1); } ");
    }
```

Figure 8: Yacc specification to control iteration sentences

4.3 Recovery of Function Names and Function Calls Graph Construction

With the attribute *caller function name* we can capture the name of defined functions in the program. This activity is shown in Figure 9.

```
function_definition
  : …….
  | declaration_specifiers  declarator  {   …   …….}
    compound_statement
    { insertFunctionInArray(&a, $<data>2.callerFunctionName,$<data>2.lineNumber, fs);
      initStack(&fs);       ……
    }
```

Figure 9: Yacc specification for recovery functions name

The function *insertFunctionInArray*(*a,name,line,list*) inserts the function with name *name*, defined in the line number *line* with a pointer to functions list *list* in the array *a*. The Figure 10 shows a snapshot of this data structure.

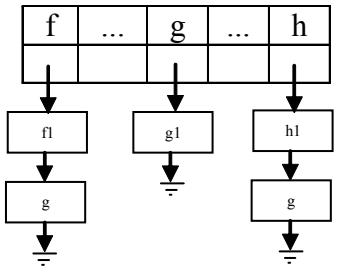

Figure 10: Data structure to store functions

Each list contains the functions called by the function stored in the array. It is important to emphasize that the data structure records recursive invocations. The list of functions is constructed when the parser finds invocation to functions in the system's source code. The Figure 11 shows the Yacc specification for this task.

```
postfix_expression
  : ……
| postfix_expression '(' ')'    {insert(&flist,$<data>1.calledFunctionName,$<data>1.lineNumber);    …}
| postfix_expression '(' argument_expression_list ')'
    {insert(&flist,$<data>1.calledFunctionName,$<data>1.lineNumber);…..}
```

Figure 11: Recovery of called functions

With all these information we can build a function calls graph which is a very important system view. We express the algorithm in pseudo code in Figure 12. The function *arrayLength*(*array*) computes the number of array's elements. The function *createNode*(*n*) creates a node with name *n*. The remaining functions are semantically understandable.

Algorithm: Function Graph
Input:
a: array of structures with the following fields: nameFunction, lineNumber, functionList.
Local Variables:
o,d: nodes. This set records the graph's nodes;
vertex: node set
relation: set of node tuples. This set holds the graph's relation
cursor: pointer
i: integer
caller, called: string
Output: G=(V,R) a function calls graph
Procedure

```
1.  vertex←∅;
2.  relation← ∅;
3.  for i=0 to arrayLength(a) do
4.    caller←a(i).nameFunction;
5.    o←createNode(name);
6.   vertex←vertex ∪ {o};
7.   cursor←a(i).functionList
8.    while (cursor≠NULL) do
9.      called← (*cursor).name;
10.     d←createNode(name)
11.     vertex←vertex ∪ {d};
12.     relation←relation ∪ {(o,d)};
13.     cursor← (*cursor).next;
14.   end while
15. end for
16.  return (vertex,relation)
```

Figure 12: Construction of Function Graph

4.4 Build of Module Communication Graph

At this moment we can identify which are the functions defined in a specific module and which are the functions called by it. With this information we know how to build a modules communication graph.We consider this system's view as an abstraction of the function calls graph. On this representation we can make abstraction on the communications. The module interaction can be described by using different functions. In other words, we can have a number of arcs between two modules. It isn't necessary to keep all theses connections because they can be recovered using the function calls graph. We only need to indicate that these modules have a connection, i.e. we need to keep only one or two connection arcs.

4.5 Other Functionalities

With the parser and its semantics actions, it is easy to recover other very important information about the system, combining the attributes, mentioned in section 4, applied with the defined techniques in section 3. In the following paragraphs we describe other functionalities provided by our approximation.

- *Return statements controls*: our techniques require that each function has at least a return sentence. When the parser finds a function without a return sentence, it must incorporate one at the end of the function. This situation can be detected using the attribute *return numbers*. If the

value of *return number* is zero then the parser's semantic actions incorporate a return sentence in the function.

- *Detection of possible programming errors*: when a function has a number of return statements greater than one, it has more that one out point. This programming practice violates structured programming rules. In this case the comprehension system reports a warning.

- *Code location*: we use the attribute *line number* to provide information about the places where the different code components will be used in the future.

- *Function types*: some constructions of ANSI-C allow to define functions without return type. The return value of theses function classes by default is an integer. The comprehension system considers that situation and it generates the appropriate code sequence at the end of the function.

Finally, we intend to complete our system, incorporating semantic actions which allow us to recover more information from the static analysis, for example, extracting information about data definitions and accesses.

Data analysis is as important as function inspection, but in our first approach we attack only the trace of functions; the study of data flow (for example, to detect which system's data is important and to choose how to show it) presents other challenges, very important but too difficult. For these reasons, we treat in this first stage functions, and in the next stage we will be concerned about data.

5. CASE STUDY: EAR, AN EVALUATOR FOR ROUTING ALGORITHMS

Some classes of routing algorithms are simple to explain because they have a very clear mathematical description. On the other hand, the implementation of this class programs is too complex. We observe this problem using EAR ("un Evaluador de Algoritmos de Ruteo") to study the performance of geometric routing algorithms. As explained in the Introduction, this was the motivation for our work.

We think that comprehension tools can help us to understand complex programs such as the geometric routing algorithms. Furthermore, the results obtained in this area about understanding routing algorithms can be generalized to other scientific or industrial complex applications.

In this initial stage, we want to prove the feasibility of applying of our techniques and then, in a future step, we will analyze its effectiveness, this is, how the information provided and views presented are actually helpful for program understanding.

Having in mind the observations made in precedent paragraphs we applied the strategy mentioned in section 3 to EAR. Initially this tool had two principal functionalities: visualization of the routing solution and complementary output; and evaluation of the routing algorithms performance. The description of these tasks can be seen in [7][13]. Now, we have incorporated the comprehension functionality.

To incorporate the inspector functions, as described in sections 3 and 4, we built a shell script that applies the parser to all system modules. Then, we compiled these modules using the EAR

compilation strategy. We identified the system's modules through the inspection of each one of directories used for it.

In the first step, we decided that the following views are important: i) output of the study program; ii) functions used to obtain the output; iii) functions' source code; iv) functions' object code. We integrated these views in four related windows (see Figure 13).

The system has navigation functions. For example, when the programmer wants to see the function's source or object code, he only has to click in the function name and the system will show it in the two other windows on the right hand side.

On the other hand, we consider that the four views have the same importance. This leads to a decision of building four windows of equal size. The product is illustrated in Figure 13. Each one of the windows is related to the others. The window on the left superior corner shows the result of the execution of the routing system's programs. In the window on the left inferior corner we can see the functions used to obtain the result. The two other windows visualize the object and source code of the selected function.

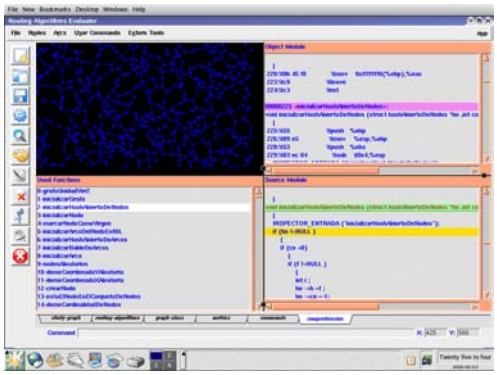

Figure 13: Different views

The importance of the views can be changed by the user through the resize windows operation. Furthermore, the user can create documentation of each function or line of code just by clicking with the mouse in the suitable line of the source or object code. The system's response is to open a text editor where he can write his comments.

At the moment we are working in the module responsible for the inter-relation between the operational or program level (information displayed at windows 2, 3 and 4) and the output or problem level (information displayed at window 1). To attain that we are implementing the connection between the function name sequence and the different components of the output of the EAR, for example nodes and arcs, graphs, etc.

6. CONCLUSION

In this paper we conceptualized Program Comprehension topic as a research field within the Software Engineering area which presents big challenges. We defined PC concepts and tools, and illustrated application areas. Then we proposed a technique for extracting dynamic and static information from source code that allows us to implement this class of programs. The referred

technique, based on code annotation, was successfully applied to our routing algorithm evaluation system, EAR. The advocated approach consists in the insertion of *inspectors* at the beginning and end of each function; also other *control functions* were inserted in the code to take care of iteration statements. The first group of statements allows us to know the functions used at execution time by the programs under study. The second class of statements gives us the possibility to reduce the output information for a better administration. To implement these techniques it is necessary to build a parser for the programming language used in EAR. With the parser we can implement our technique, and construct other system views (like the *function calls graph* and the *modules communication graph*). In a similar way, we can detect possible programming and structural errors. Applying this approach to the routing system (see section 5), it permitted us to think about the information visualization, which is another important characteristic of comprehension software. In this sense, we built four views and we assigned importance to each one. We thought that our strategies of visualization and information extraction could be used in many different systems.

The principal advantage provided by our approach, when compared with other professional tools for debugging, tracing, cross referencing, etc., is that our system creates the annotations automatically. On the other hand, the presentation of the different views and the assignment of importance to each window is another very important a characteristic, normally not provided by other tools.

As future work, we intent to complete our annotation scheme with strategies to control the recursive functions, and other characteristics of the programming languages. Besides that, we also intend to develop animation strategies from system's behavior and for each action, as well as build explanations to decorate the algorithms output. Finally, we want to assess our tool collecting experimental indicators about its use for program understanding.

7 REFERENCES

[1] Berón, M., Gagliardi, O., Hernández Peñalver, G. *Evaluación de Métricas en Redes de Computadoras*. Expuesto y publicado en el Congreso Argentino de Ciencias de la Computación (CACIC 2003), realizado en la Universidad de La Plata. La Plata. 2003.

[2] Berón, M., Gagliardi, O., Flores, S. *Ruteo en Redes Inalámbricas*. Expuesto y publicado en el Congreso Argentino de Ciencias de la Computación (CACIC 2002), realizado en la Universidad de Buenos Aires. 2002.

[3] Berón, M., Gagliardi, O., Hernández Peñalver, G. *"Evaluación de Algoritmos de Ruteo de Paquetes en Redes de Computadoras"*. Expuesto y publicado en el Workshop de Investigadores en Ciencias de la Computación (WICC 2004), realizado en la Universidad Nacional del Comahue. Año: 2004.

[4] Berón, M., Gagliardi, O., Hernández Peñalver, G. *"Evaluación de Algoritmos de Ruteo en Redes de Computadoras"*. Expuesto y publicado en el V Workshop de Investigadores en Ciencias de la Computación (WICC 2003), realizado en la Universidad Nacional del Centro. Tandil. 2003.

[5] Berón, M., Grosso, A., Gonzales, A. Printista, M., Maldocena, P., Ordoñes, G., Molina, S., Apolloni, R.. *"Una Aproximación hacia el estudio de los Sistemas de Computación"*. Expuesto y publicado en el V Workshop de Investigadores en Ciencias de la Computación (WICC 2003), realizado en la Universidad Nacional del Centro. Tandil. 2003.

[6] Berón, M., Hernández Peñalver, G., Gagliardi, O. *"Factibilidad de Uso del Ruteo Voraz en los Grafos de Gabriel, Vecindad Relativa y Triangulaciones"*. Expuesto y publicado en el

Congreso Argentino de Ciencias de la Computación (CACIC 2004), realizado en la Universidad Nacional de la Mantanza. 2004.

[7] Berón, M; Henriques, P; Varanda, M; Uzal, R. "Comprensión de Algoritmos de Ruteo". Expuesto y publicado en la 32a Conferencia Latinoamericana de Informática (CLEI 2006), realizado en Santiago de Chile. 2006.

[8] Berón, M; Henriques, P; Varanda, M; Uzal, R. *"Herramientas para la Comprensión de Programas"*. Expuesto y publicado en el Workshop de Investigadores en Ciencias de la Computación (WICC 2006), realizado en la Universidad de Morón. Buenos Aires. 2006.

[9] Berón, M; Hernádez Peñalver, G; Gagliardi, O. "*Un Evaluador de Algoritmos de Ruteo"*. Master Thesis in Software Engineering. Universidad Nacional de San Luís. 2005.

[10] Brooks,R. *"Using a behavioral theory of program comprehension in software engineering"*. In Proceedings of the 3rd international conference on Software Engineering (ICSE '78), pages: 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[11] Grosso, A., Riesco, D., Berón, M. *"Una Herramienta para la Ingeniería Inversa de Sistemas Escritos en Lenguaje C, Bajo Linux"*. Comunicación, Expuesta y Publicada en el Acta de Resúmenes del Congreso Argentino de Ciencias de la Computación (CACIC 2002), realizado en la Universidad de Buenos Aires. 2002.

[12] Hamou-Lhadj, A. Lethbridge. *"A Survey of Trace Exploration Tools and Techniques"*. Proceedings of the conference of the Centre for Advanced Studies on Collaborative research, pages: 42-55. 2004.

[13] Hernández Peñalver, G., Berón, M., Gagliardi, O. *"Ruteo Geométrico Aplicado a las Redes de Computadoras"*. Expuesto y publicado en el Workshop de Investigadores en Ciencias de la Computación (WICC05), realizado Universidad Nacional de Río Cuarto. 2005.

[14] Linter, R; Michand, J; Storey, M; Wu, X. *"Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse"*. ACM Symposium on Software Visualizaton, pages: 47-56. 2003.

[15] Mayrhauser, A; Vans, M. "*Program Comprehension During Software Maintenance and Evolution*". Computer, vol. 28, no. 8, pages: 44-55. 1995.

[16] Moreno, A., Myller, N., Sutinen, E., and M. Ben-Ari. (2004b). *"Visualizing Programs with Jeliot 3"*. Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI 2004), pages: 373–376.

[17] Oliveira, E; Henriques, P; Varanda, M. *"Características de um Sistema de Visualização para Compreensão de Aplicações Web através de Inspecção de Software e baseadas em Modelos Cognitivos"*. Master Thesis in Software Engineering. Universidade do Minho. 2006.

[18] Pacione, M; Roper, M; Wood, M. *"A Comparative Evaluation of Dynamic Visualization Tools"*. Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), pages: 80-89. IEEE. 2003.

[19] Sajaniemi, J. *"Program Comprehension through Multiple Simultaneous Views: A Session with VinEd"*, page: 99. IEEE. 2000.

[20] Storey, D; Fracchia, F; Müller, H. "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization". Proceedings of the 5th International Workshop on Program Comprehension (WPC '97), pages: 17-28, May 1997.